HW5

Yuzhen Chen

December 20, 2024

1 Section1: Pix2Pix

- Task 1.1: Dataloading

Q1 : Implement the Edges2Image class and fill in the TODOs in that cell.

A1 :

```
    class Edges2Image(Dataset):
  def __init__(self, root_dir, split='train', transform=None):
    """
    Args:
        root_dir: the directory of the dataset
        split: "train" or "val"
        transform: pytorch transformations.
    """

    self.transform = transform

    self.files = glob.glob(os.path.join(root_dir, split, '*.jpg'))

  def __len__(self):
    return len(self.files)

  def __getitem__(self, idx):
    img = Image.open(self.files[idx])
    img = np.asarray(img)
    if self.transform:
        img = self.transform(img)
    return img

transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])

###########################################################################
# TODO: Construct the dataloader                                          #
# For the train_loader, please use a batch size of 4 and set shuffle True #
# For the val_loader, please use a batch size of 5 and set shuffle False  #
# Hint: You'll need to create instances of the class above, name them as  #
# tr_dt and te_dt. The dataloaders should be named as train_loader and    #
# test_loader. You also need to include transform in your class           #
#instances                                                                #
###########################################################################

tr_dt = Edges2Image(root_dir="/home/yuzhen/Desktop/EECS442/hw5_updated/hw5-easy-release/mini-edges2shoes",split="train",transform=transform)
te_dt = Edges2Image(root_dir="/home/yuzhen/Desktop/EECS442/hw5_updated/hw5-easy-release/mini-edges2shoes",split="val",transform=transform)

train_loader = DataLoader(dataset=tr_dt,batch_size = 4,shuffle=True)
test_loader = DataLoader(dataset=te_dt,batch_size = 5,shuffle=False)

###########################################################################
#                              END OF YOUR CODE                           #
###########################################################################

# Make sure that you have 1,000 training images and 100 testing images before moving on
```

```
print('Number of training images {}, number of testing images {}'.format(len(tr_dt), len(te_dt)))


```

- Task 2: Training Pix2Pix

  Q1 : Please set up G_optimizer and D_optimizer in the train function.

  A1 :

```
            def train(G, D, num_epochs = 20):
  hist_D_losses = []
  hist_G_losses = []
  hist_G_L1_losses = []
  ############################################################################
  # TODO: Add Adam optimizer to generator and discriminator                  #
  # You will use lr=0.0002, beta=0.5, beta2=0.999                            #
  ############################################################################

  lr = 0.0002
  beta1 = 0.5
  beta2 = 0.999
  G_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(beta1, beta2))
  D_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(beta1, beta2))

  ############################################################################
  #                               END OF YOUR CODE                           #
  ############################################################################

```

  Q2 : Implement the code for the function train as instructed by the notebook.

  A2 :

```
           # Hint: you could use following loss to complete following function
BCE_loss = nn.BCELoss().cuda()
L1_loss = nn.L1Loss().cuda()

def train(G, D, num_epochs = 20):
  hist_D_losses = []
  hist_G_losses = []
  hist_G_L1_losses = []
  ############################################################################
  # TODO: Add Adam optimizer to generator and discriminator                  #
  # You will use lr=0.0002, beta=0.5, beta2=0.999                            #
  ############################################################################

  lr = 0.0002
  beta1 = 0.5
  beta2 = 0.999
  G_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(beta1, beta2))
  D_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(beta1, beta2))

  ############################################################################
  #                               END OF YOUR CODE                           #
  ############################################################################

  print('training start!')
  start_time = time.time()
  for epoch in range(num_epochs):
    print('Start training epoch %d' % (epoch + 1))
    D_losses = []
    G_losses = []
    epoch_start_time = time.time()
    num_iter = 0
    for x_ in train_loader:
      y_ = x_[:, :, :, img_size:]
      x_ = x_[:, :, :, 0:img_size]

      x_, y_ = x_.cuda(), y_.cuda()
      ############################################################################
      # TODO: Implement training code for the discriminator.                     #
      # Recall that the loss is the mean of the loss for real images and fake    #
      # images, and made by some calculations with zeros and ones                #
```

```
      # We have defined the BCE_loss, which you might would like to use.        #
      #                                                                          #
      # NOTE: While training the Discriminator, the output of the generator     #
      # must be detached from the computational graph. Refer to the method      #
      # torch.Tensor.detach()                                                    #
      ############################################################################

      #reset the optimizer
      D_optimizer.zero_grad()

      # train on real data
      x_y_ = torch.hstack((x_, y_))
      D_real_preds = D(x_y_)
      D_real_preds_shape = D_real_preds.shape
      the_real_label = torch.ones(D_real_preds_shape,device = device)

      D_real_loss = BCE_loss(D_real_preds, the_real_label)

      # train on fake data
      fake_data = G(x_).detach()
      D_fake_preds = D(torch.hstack((x_, fake_data)))
      D_fake_preds_shape = D_fake_preds.shape
      the_fake_label = torch.zeros(D_fake_preds_shape,device = device)
      D_fake_loss = BCE_loss(D_fake_preds, the_fake_label)

      loss_D = (D_real_loss + D_fake_loss) / 2

      loss_D.backward()
      D_optimizer.step()

      ############################################################################
      #                              END OF YOUR CODE                            #
      ############################################################################

      ############################################################################
      # TODO: Implement training code for the Generator.                         #
      ############################################################################

      # 1. Train the generator
      # 2. Append the losses to the lists 'hist_G_L1_losses' and 'hist_D_losses'
      #  (Only append the data to the list, not the complete tensor, refer
      #  torch.Tensor.item()).

      G_optimizer.zero_grad()


      generator_data = G(x_)
      x_gen_data = torch.hstack((x_, generator_data))
      prediction = D(x_gen_data)
      G_L1_loss = L1_loss(generator_data, y_)
      lambda_L1 = 100
      loss_G = BCE_loss(prediction,the_real_label) + lambda_L1 * G_L1_loss


      loss_G.backward()
      G_optimizer.step()
      ############################################################################
      #                              END OF YOUR CODE                            #
      ############################################################################

      D_losses.append(loss_D.detach().item())
      hist_D_losses.append(loss_D.detach().item())
      G_losses.append(loss_G)

      hist_G_losses.append(loss_G.detach().item())
      hist_G_L1_losses.append(G_L1_loss.detach().item())
      num_iter += 1



    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time

    print('[%d/%d] - using time: %.2f seconds' % ((epoch + 1), num_epochs, per_epoch_ptime))
    print('loss of discriminator D: %.3f' % (torch.mean(torch.FloatTensor(D_losses))))
    print('loss of generator G: %.3f' % (torch.mean(torch.FloatTensor(G_losses))))
```

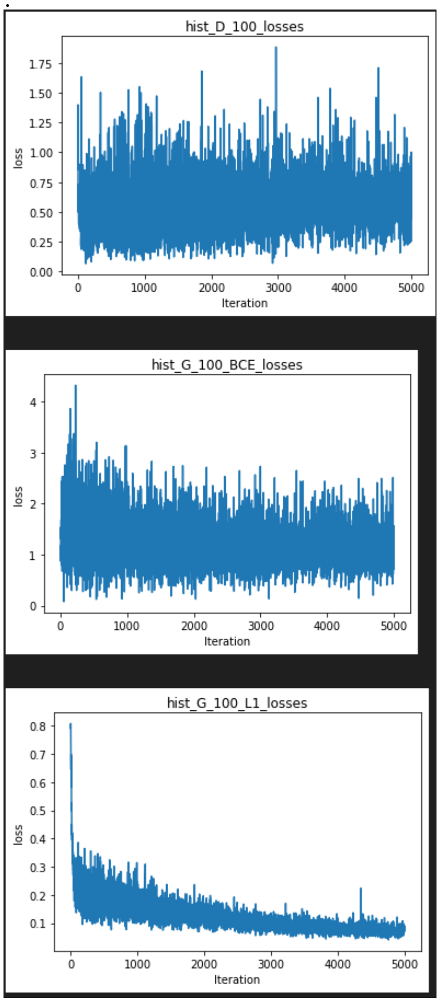
```python
    if epoch == 0 or (epoch + 1) % 5 == 0:
      with torch.no_grad():
        show_result(G, fixed_x_, fixed_y_, (epoch+1))

  end_time = time.time()
  total_ptime = end_time - start_time

  return hist_D_losses, hist_G_losses, hist_G_L1_losses
```

Q3 : In your report, include these plots.

A3 :



2 Section 2: Diffusion Models

- : Task 3.1: Implement the method get_named_beta_schedule with linear and cosine schedules.

  Q1 :

  A1 :

```python
def get_named_beta_schedule(schedule_name, num_diffusion_timesteps, beta_min=0.0001, beta_max=0.02):
    """
    Get a pre-defined beta schedule for the given name.

    Args:
        schedule_name: str, name of the variance schedule, 'linear' or 'cosine'
        num_diffusion_timesteps: int, number of the entire diffusion timesteps
        beta_min: float, minimum value of beta
        beta_max: float, maximum value of beta

    Returns:
```

```
        betas: np.ndarray, a 1-d array of size num_diffusion_timesteps, contains all the beta for each timestep

    """
    betas = [0]
    if schedule_name == "linear":
        ########## START TODO ##########
        # Implement the linear schedule
        # Uniformly divide the [beta_min, beta_max) to num_diffusion_timesteps values.

        betas = np.arange(beta_min, beta_max, (beta_max - beta_min) / num_diffusion_timesteps)
        ########## END TODO ##########
    elif schedule_name == "cosine":
        ########## START TODO ##########
        # Implement the cosine schedule
        # Assume s = 0.008 and beta_clip=0.999
        s = 0.008
        beta_clip = 0.999

        # betas = np.zeros(num_diffusion_timesteps)
        old_alpha = 1
        normalize_factor = math.cos((s/(1+s))*(math.pi/2))**2
        T = num_diffusion_timesteps
        for t in range(1,T):
            alpha = math.cos((((t/T)+s)/(1+s))*(math.pi/2))**2 / normalize_factor
            min_beta = min(beta_clip,1-(alpha/old_alpha))
            betas.append(min_beta)
            #update the new alpha
            old_alpha = alpha
        del betas[0]
        betas.append(0.999)

        ########## End TODO ##########


        # Add a value to the end of betas

    else:
        raise NotImplementedError(f"unknown beta schedule: {schedule_name}")

    return betas

```

- : Task 4.1: p_sample and p_sample_loop of class DDPMDiffusion

  Q1 : Fill the TODO sections of the class DDPMDiffusion of the file guided_diffusion/simple_diffusion.py. Complete the methods p_sample and _sample_loop.

  A1 :

```
            ########### START TODO  ###########
        # Calculate the values of alpha
        # Also we will need the cumulated product of alpha.
        # And during sampling we need the value of cumulated product of alpha from
        # previous or next timestep.
        self.alphas = 1 - self.betas
        self.alphas_cumprod = np.cumprod(self.alphas) # cumpulated product of alphas
        self.alphas_cumprod_prev =np.concatenate(([1.0], self.alphas_cumprod[:-1]))
        self.alphas_cumprod_next = np.concatenate((self.alphas_cumprod[1:], [0.0]))

        ########### END TODO  ###########

```

```
            def p_sample_loop(self,
                      model,
                      x_start,
                      record,
                      save_root,
                      measurement=None,
                      measurement_cond_fn=None,
                      uncond=False):
        """
        The function used for sampling from noise.
```

```
        Args:
            model: nn.Module, the pretrained model that is used to predict the score and variance
            x_start: torch.Tensor, random noise input
            measurement: torch.Tensor, our corrupted observation
            measurement_cond_fn: conditional function used to perform conditional sampling, is None for unconditional sampling
            record: Bool, save intermediate results if True
            save_root: str, root of the directoy to save the results
            uncond: Bool, perform unconditional sampling if True, else perform conditional sampling
        """
        if not uncond:
            assert measurement is not None and measurement_cond_fn is not None, \
                "measurement and measurement conditional function is required for conditional sampling"

        img = x_start   # start from random noise
        device = x_start.device

        ############ Start TODO #############
        # Implement the sample loop
        # Call p_sample for every iteration
        # It requires only one line of code implementation here

        pbar = tqdm(list(range(self.num_timesteps))[::-1])
        for idx in pbar:
            time = torch.tensor([idx] * img.shape[0], device=device)

            img = self.p_sample(model,img,time)["x_t_minus_1"]

            img = img.detach_()

        ##########################################
            if record:
                if idx % 10 == 0:
                    file_path = os.path.join(save_root, f"progress/x_{str(idx).zfill(4)}.png")
                    plt.imsave(file_path, clear_color(img))

        return img

```

```
             def p_sample(self, model, x, t):
        """
        Posterior sampling process, when given the model, x_t and timestep t, it returns predicted
        x_0 and x_t_minus_1

        We have already provide you with the function to get the log of the variance.
        Use self.mean_processor.get_mean_and_xstart(var_values, t), where var_values is
        the 3:6 channels of the direct output of the model.
        example usage: log_variance = self.var_processor.get_variance(var_values, t)

        You can also use the helper function extract_and_expand() to extract the value
        corresponding to timestep and expand it to the save size as the target for broadcast.
        example usage: coef1 = extract_and_expand(self.posterior_mean_coef1, t, x_start)

        Args:
            model: nn.Module, the UNet model, you can call model(x, t) to get the output tensor with size (B, 6, H, W)
            x: torch.Tensor, shape (1, 3, H, W), x_t
            t: torch.Tenosr, shape (1,), timestep

        Returns:
            output_dict: dict, contains predicted x_t_minus_1 and x_0
        """
        #####Start TODO#####
        ##### Get the predicted score and variance of the pretrained model #####
        pred_noise = model(x,t)[:,:3]
        var_values = model(x,t)[:,3:6]
        ##### End TODO #####

        log_variance = self.var_processor.get_variance(var_values, t)   # get the log  of variance

        #####  Start TODO   #####
```

```
        ##### get predicted x_0 and x_t_minus_1  #####
        ##### don't forget to add noise for all the steps, except for the last one  #####
        exp_log_variance = torch.exp(log_variance)
        sigma = torch.sqrt(exp_log_variance)

        coef1 = extract_and_expand(self.alphas, t, x).cuda()
        coef2 = extract_and_expand(self.alphas_cumprod, t, x).cuda()

        if t > 0:
            z = torch.randn(x.size()).cuda() #test
        else:
            z = torch.zeros_like(x)



        first_part = z * sigma
        numerator = x-((1-coef1) / torch.sqrt(1-coef2)) * pred_noise
        denominator = torch.sqrt(coef1)
        second_part = numerator / denominator


        x_t_minus_1 = first_part + second_part

        #####  End TODO   #####

        assert x_t_minus_1.shape == log_variance.shape == x.shape

        output_dict = {'x_t_minus_1': x_t_minus_1}
        return output_dict

```

- : Task 4.2:Sampled Image

  Q1 :

  A1 :

- : Task 5.1:p_sample of class DDIMDiffusion

  Q1 :

  A1 :

```
             def p_sample(self, model, x, t, eta=0.0):
        ##############################################
        #####   TODO    #####
        ##### Get the predicted score and variance of the pretrained model #####
        ##### Don't forget to use _scale_timesteps to scale the timestep for calling the model
    prediction.
        ##### You don't need to scale the timestep for further computations of x_t_minus_1.
        ##### NOTE: Since this version of the model learns the variance along with the score
    function,
        ##### the output of the model would have double the number of channels as that of the
    input.
        ##### So assign the predicted score and variance values to the variables below.Refer to
        ##### torch.split method.
        ##############################################
```

```python
        #####  Start TODO   #####

        scaled_t = self._scale_timesteps(t)
        mode_output = model(x,scaled_t)
        pred_noise = mode_output[:, :x.shape[1]]
        ##### End TODO #####

        model_mean, pred_xstart = self.mean_processor.get_mean_and_xstart(x, t, pred_noise)

        #####   TODO    #####
        ##### Step 1: Implement the variance parameter 'sigma' for DDIM sampling.   #####
        ##### Step 2: Imeplemnt x_t_minus_1 using the pred_xstart. Don't forget     #####
        #####  to add noise for all the steps, except for the t=0.                  #####
        #####                                                                       #####
        ##### You may use the function 'extract_and_expand' to expand the timestep  #####
        ##### variable 't' to the input's shape.                                    #####
        ##### Assign them to the variables x_t_minus_1.                             #####

        #####  Start TODO   #####
        a_t = extract_and_expand(self.alphas_cumprod,t,x).cuda()
        a_t_minus_1 = extract_and_expand(self.alphas_cumprod_prev,t,x).cuda()
        if t > 0:
            z = torch.randn_like(x).cuda()
        else:
            z = torch.zeros_like(x)
        sigma = eta*torch.sqrt(torch.sqrt((1-a_t_minus_1)/(1-a_t))*torch.sqrt(1-(a_t)/(a_t_minus_1)))

        predicted_x0 = torch.sqrt(a_t_minus_1) * ((x-torch.sqrt(1-a_t)*pred_noise)/(torch.sqrt(a_t)))
        direction_xt = (torch.sqrt(1-a_t_minus_1-(sigma**2))*pred_noise)
        x_t_minus_1 =  predicted_x0 + direction_xt

        if t > 0:
            x_t_minus_1 += sigma * z
        ##### End TODO #####

        return {"x_t_minus_1": x_t_minus_1, "pred_xstart": pred_xstart}
        ##############################################


    def predict_eps_from_x_start(self, x_t, t, pred_xstart):
        coef1 = extract_and_expand(self.sqrt_recip_alphas_cumprod, t, x_t)
        coef2 = extract_and_expand(self.sqrt_recipm1_alphas_cumprod, t, x_t)
        return (coef1 * x_t - pred_xstart) / coef2
```

- : Task 5.2:Sampled Image

  Q1 :

  A1 :

- : Task 7.1:PosteriorSampling

  Q1 :

  A1 :

```python
            def conditioning(self, x_i, x_t_minus_one, x_0_hat, measurement, **kwargs):
        """
        The conditioning function as shown in line 7

        Args:
            x_i: torch.Tensor, x_i
            x_t, torch.Tensor, x_t_minus_1 prime
            x_0_hat: torch.Tensor, predicted x_0
            measurement: torch.Tensor, y, the corrputed image
        """
        # norm_grad, norm = self.grad_and_value(x_prev=x_prev, x_0_hat=x_0_hat, measurement=measurement, **kwargs)
        ############    Start TODO   ###########
        ###### Inplentment the conditional sampling in line 7 #######
        ###### A(x_0_hat) is already provided to you as A ########
        ###### Also torch.autograd.grad() is provided to you to calculate the gredient of the
        ###### norm term with respect to x_i, you can check https://pytorch.org/docs/stable/generated/torch.autograd.grad.html#torch.autograd.grad
        ###### for its detailed usage. You only need to specify the outputs and inputs here.
        # A = self.operator.forward(x_0_hat, **kwargs)
        A = self.operator.forward(x_0_hat,**kwargs)

        difference = measurement - A
        norm = torch.sqrt(torch.sum(difference ** 2))
        diff_output = norm  # outputs of the differentiated function
        diff_input = x_i   # Inputs w.r.t. which the gradient will be returned

        ## TODO: Don't delete this line, you will use this
        norm_grad = torch.autograd.grad(outputs=diff_output, inputs=diff_input)[0]
        new_x_t_minus_one = x_t_minus_one - norm_grad
        ############    END TODO   ###########
        return new_x_t_minus_one

```

- : Task 7.2: Inpainted Image

  Q1 :

  A1 :